REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
22/08/2013	Final Technical Report	01/11/2012 - 31/05/2013

4. TITLE AND SUBTITLE

100K+ Core Challenge for Comprehensive Computing at Scale
Deliverable 2
Final Technical Report

5a. CONTRACT NUMBER
GS04T09DBC0017

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Ziegeler, Sean; Pettey, Lucas; Koop, Matt

5d. PROJECT NUMBER
PP-CCM-KY04-013-P3

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

High Performance Technologies Inc.
11440 Commerce Park Drive
Suite 600
Reston, VA 20191

8. PERFORMING ORGANIZATION REPORT NUMBER

PP-CCM-KY04-013-P3-D2

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Department of Defense
High Performance Computing Modernization Program Office
10501 Furnace Road
Suite 101
Lorton, VA 22079

10. SPONSOR/MONITOR'S ACRONYM(S)

HPCMP, PETTT

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution Statement A: Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

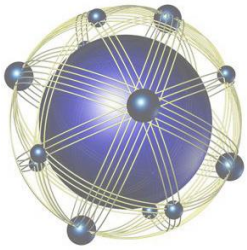
This document is the Final Technical Report for the PETTT Pre-Planned Project PP-CCM-KY04-013-P3, 100K+ Core Challenge for Comprehensive Computing at Scale. This report presents advanced Input/Output (I/O) and co-processing analysis routines as utilized by a computational model scaling to large numbers of compute cores. The report also includes the background on the model used as a case study, I/O routine overviews, co-processing routine overviews, and timings and performance results of these routines.

namics for simulating liquids crystals. We further
ified algorithm for imposing irrotational constraints on the simulation cell dynamics, as well as
gues, with the goal of increasing numerical stability for simulations that impose a constant
ciples molecular dynamics simulations of liquid alloys.
namics for simulating liquids crystals.
simplified algorithm for imposing irrotational constraints on the simulation cell dynamics,
olleagues, with the goal of increasing numerical stability for simulations that impose a
principles molecular dynamics simulations of liquid alloys.

15. SUBJECT TERMS

PETTT, HPC, HPCMP, I/O, ADIOS, Visualization, Co-processing, Bose-Einstein Condensates, Lustre, MPI-IO

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
Unclassified					Andrew Mark
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	UU	14	**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified	Unclassified	Unclassified			703-812-4453



High Performance Technologies Inc.
11440 Commerce Park Drive, Suite 600
Reston, VA 20191

100K+ Core Challenge for Comprehensive Computing at Scale

PP-CCM-KY04-013-P3
Deliverable 02

Final Technical Report

Submitted by
Sean Ziegeler, Lucas Pettey, & Matt Koop
22 August 2013

for
User Productivity Enhancement, Technology Transfer, and Training (PETTT) Program

High Performance Computing Modernization Program (HPCMP)

Contract No.: GS04T09DBC0017

Government Users/Sponsors/Partners: Bob Maier (ARMY), George Vahala, Linda Vahala, Min Soe, Jeff Yepez

TABLE OF CONTENTS

1. OVERVIEW

This Final Technical Report documents the completion of PETTT Pre-Planned Project PP-CCM-KY04-013-P3.

This report presents advanced Input/Output (I/O) and co-processing analysis routines as utilized by a computational model scaling to large numbers of compute cores. The report also includes the background on the model used as a case study, I/O routine overviews, co-processing routine overviews, and timings and performance results of these routines.

1.1 Introduction

The largest system in the DoD HPCMP, Garnet, has been recently made available at the ERDC DSRC. This system is a combination of three smaller Cray XE6 systems and now consists of 150,592 compute cores. This Pre-Planned Effort (PPE) conducted a test case study of high performance I/O with George Vahala's Bose-Einstein Condensate (BEC) simulation code using up to 150,000 cores.

The BEC code already included a number of I/O methods, one of which sustained one of the fastest output performance rates on DoE supercomputers. The data produced by these I/O methods in the model is used for visualization and analysis to explore the simulated properties of BEC materials. This PPE installed the necessary I/O and co-processing libraries to support the code on Garnet, modified it when necessary to use these libraries, and then compared the performance between the libraries. The reason for comparing the various I/O and co-processing methods is to determine the most effective method for Garnet and similar large-scale HPC systems. Some of the methods should be theoretically faster than others, but the hardware configuration may produce different results that would only be obtained by explicit testing. The overarching goal of this PPE is to produce best practices for large-scale computing, primarily for speeding up applications with significant output data volume with the need to analyze that data for scientific discovery.

1.2 User Code Description

A group led by Principal Investigator (PI) George Vahala developed a code to simulate Bose-Einstein Condensates (BECs) using some unitary qubit ideas of Jeffrey Yepez [2, 3]. BECs are a unique state of matter existing at extremely cold temperatures and have properties linked to superfluids and superconductivity. They are of interest to the DoD for next-generation, ultra-sensitive sensor technology as well as use in the design of quantum computers.

The code in this study is simply named after the matter which it simulates - BEC. BEC has advanced the field of Bose-Einstein condensates and may lead to new discoveries in this area of theoretical physics. The method it uses is based on a non-linear quantum lattice gas model providing an *ab initio* representation of superfluid dynamics implemented using a quantum logic algorithm. Vahala has run the code on nearly every HPCMP system, in many cases as part of Pioneer Projects and Capability Applications Projects (CAP), using all cores on the systems before the system is available to general users. In addition, the BEC code has been run on Department of Energy (DoE) systems at leadership-class scales (over 200,000 cores). The BEC

code had already run on Garnet before the integration, but it did have to be recompiled. All dependent libraries also had to be recompiled and relinked.

1.3 Approach

The overall plan was to modify the code to utilize various I/O and co-processing libraries and methods, add benchmark timing calls for each method, and then execute the code with each method. For each benchmark run, the code was switched (using simple compile-time flags) to use only that method. These methods are shown in Figure 1 and described in Section 1.3.1.

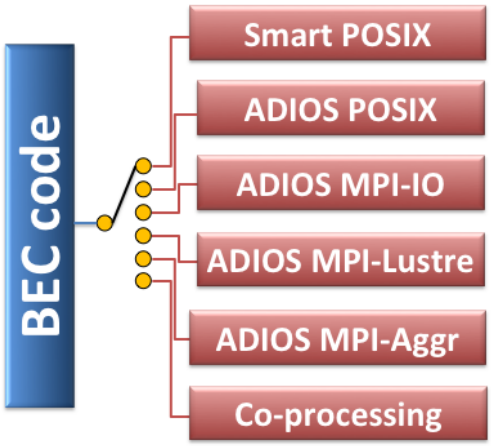


Figure 1. Modes of execution for I/O and co-processing methods

1.3.1 I/O Libraries

A "library" is a collection of software routines that can be utilized by source code in a modular fashion. This is done for convenience and is good design practice for software in general. I/O libraries implement subroutines that typically make I/O easier, better, and/or faster than writing I/O code from scratch. This study included two I/O libraries, Smart POSIX and ADIOS.

1.3.1.1 Smart POSIX

The Smart POSIX library provides a method that uses traditional I/O system calls, but these are modified to limit the number of parallel tasks that simultaneously access the file system at once. The functionality is encapsulated in a C library and also provides Fortran bindings and include files. All options and data structures are specified in the user's code, including simple array bounds, metadata, an MPI communicator, and the array itself. This can be repeated for multiple arrays. The user can set the maximum number of concurrent I/O operations, and the library will limit the concurrent accesses by passing a series of tokens to the MPI tasks. Each task writes a separate data file grouped together under a subdirectory with a separate file containing metadata. These files are in the ParaView Parallel VTK Image (PVTI) format (here, "image" refers to a three-dimensional image that represents a Cartesian volumetric data set). This method was previously implemented through a PETTT Reactionary Assistance effort performed years ago by a member of this PPE team.

1.3.1.2 ADIOS

The four other I/O methods in this study are provided by the ADaptable Input/Output System (ADIOS), which is designed as a primary interface layer atop other I/O libraries or transports. This allows a user to swap-out I/O methods with very little effort to tailor the I/O to the hardware of the HPC system. ADIOS has been shown to perform very well on leadership-class HPC systems, sometimes improving I/O performance by more than 1000-times over other I/O methods in parallel codes [1]. All of the ADIOS transport methods use a consistent metadata format known as "BP." Thus, any tool using ADIOS may switch between I/O methods without creating format incompatibilities when attempting to read the output data later.

ADIOS POSIX is very similar to the Smart POSIX approach. As such, it writes a single file per MPI task, grouped under a single directory and intelligently manages concurrent I/O accesses. However, one major difference between Smart POSIX and ADIOS POSIX is that the latter aggressively buffers the data for better streaming performance.

ADIOS MPI-IO uses the Message Passing Interface Input/Output (MPI-IO) library. MPI-IO is included as part of the MPI-2 and later standards and is commonly used for parallel file system I/O. MPI-IO facilitates the efficient use of a single file for multiple tasks, making data management much easier. The ADIOS MPI-IO transport method also utilizes serialized MPI-IO open calls and other timing delays to reduce the load on the file system metadata server.

ADIOS MPI-Lustre is based on the MPI-IO transport method. However, MPI-Lustre adds stripe alignment to improve write performance. ADIOS polls the file system via Lustreapi (if available) and uses the information to align the writing tasks' data to the Lustre stripes. This improves the parallelization of the write operations. Similar to the ADIOS MPI-IO transport, there is only one output file for all tasks.

ADIOS MPI-Aggregate (MPI-Aggr) is based on the MPI-Lustre transport method. The original design was intended to accommodate adaptive mesh codes, but it was found to work well for almost any grid or data scheme. MPI-Aggr maximizes performance for large-scale codes, i.e., more than 10,000 cores, according to the ADIOS manual [1]. It does this by (1) aggregating data from multiple MPI tasks into large chunks on fewer tasks, reducing the number of I/O requests; (2) performing non-blocking file opens and initialization; and (3) writing one file per Lustre *object storage target* (OST) to reduce communication and wide striping overhead. If Lustreapi is available on the system, ADIOS will retrieve the necessary information automatically.

1.3.2 Co-Processing Libraries

Simulation on HPC systems can usually be divided into 3 steps: pre-processing, model execution, and post-processing. Post-processing refers to some sort of analysis and/or visualization of the model output. Often, this means having the model write out data (perhaps using one of the I/O methods above), then upon completion of the model execution, launching a separate job to read that data back in and perform the analysis. The I/O steps in this process can often be very slow relative to the actual model execution and post-processing steps.

Co-processing is the coupling of the model execution and post-processing steps such that they occur simultaneously. This is often accomplished through the use of a library and adding

instrumentation calls in the model code to provide the data to the co-processing library's subroutines. In many ways, adding instrumentation to a code is very similar to using an I/O library.

Co-processing eliminates the large amount of file system I/O needed to convey data between the model and post-processing. Often, this can speed up by an order of magnitude the portions of model execution that would normally be performing I/O. An additional advantage, often overlooked, is that the post-processing step is correspondingly faster. Perhaps most importantly, given that the low I/O bandwidth necessitates writing some infrequent subset of model time steps, is that co-processing allows one to analyze model results at a much higher temporal frequency. There is potentially significant information that can be gleaned from analyzing and/or visualizing at or closer to every time step of the model [4].

1.3.2.1 ParaView Catalyst

ParaView is a standard, open-source visualization software package. It provides a variety of visualization options, operates in parallel on small laptops, and scales up to HPC systems. It is most often used for traditional post-processing, by opening data files, setting up visualization options, and rendering visual results.

Catalyst is a subset of the visualization and rendering core extracted from the rest of the ParaView code base, but with additional subroutines to allow instrumentation of user code. Users can utilize the ParaView GUI to configure their desired visualization features and options, then save that configuration in a Python script. Catalyst will load that Python script at initialization, when the model begins to run, and use it to generate extractions and render images in the same configuration as specified. The final result could be an extracted data subset, such as isosurfaces or cross-sections) and/or rendered images. If an extracted data subset is written to files, we refer to this as the Catalyst "Extract" option. The advantage to this approach is that the data subsets may be re-loaded by interactive visualization software more quickly than the original data. It may also be inspected in any way (e.g., by changing viewing angles, color maps, data queries, etc.). If the data or results are rendered to an image, we refer to this as Catalyst "Render." It is generally much faster than Extract because I/O is nearly eliminated, but the resultant image cannot be manipulated after the fact. If render options need to be changed, the model must be re-run, so it is necessary to have some a priori knowledge of the render options.

1.3.2.2 QIso

QIso is a co-processing library originally created as part of PETTT Reactionary Assistance efforts. Its original function was to generate isosurfaces (contour surfaces of a constant value) independently from each MPI task and save an image. A separate compositor stitched the images together. In this PPE, it was modified to do parallel compositing such that the final result is a single image (per time step). It has very limited capability compared to Catalyst (e.g., only a Render option), but it is also small, low overhead, and very fast.

2. ACCOMPLISHMENTS

2.1 Smart POSIX Modification

The Smart POSIX library is based on an older PETTT project known as JPEG2000-3D for High Performance Computing (J3D4HPC). Eventually the JPEG2000 compression was dropped because the computation required to compress the data was not worth the storage savings. However, one feature that persisted was a data reduction technique that quantized (scaled) the floating-point arrays to single bytes (versus the full 4-byte or 8-byte values). ADIOS does not natively support this, but we could have added a layer to perform the quantization before using ADIOS. However, as the users of BEC have scaled to larger grids, they have found that they need the full precision of floating-point values to properly query and visualize the output data sets. So we chose to add the floating-point output support to the Smart POSIX library. Users of the library may now choose between 1-byte, 2-byte, 4-byte, and no quantization, using a single library call, for various levels of fidelity.

2.2 ADIOS Installation on Garnet

ADIOS 1.5.0 was installed on Garnet as part of this PPE. There are installations for each of the compilers. To utilize these, load the appropriate compiler environment, e.g.,

module load PrgEnv-{cray|gnu|intel|pgi}.

Then load the adios module, as follows:

module load adios

ADIOS on Garnet is configured to support the NULL, POSIX, MPI-IO, MPI-Lustre, and MPI-Aggr transport methods. It is also linked to the NetCDF and HDF5 libraries so that the *bin/* directory of the above installations also includes conversion utilities such as *bp2h5* and *bp2ncd*, which will convert ADIOS' BP format files to HDF5 and NetCDF files, respectively. Consult the ADIOS manual [1] for instructions on how to select transport methods, use conversion utilities, and various other features of ADIOS.

2.3 BEC Modifications and Instrumentation

BEC already included options to use both the Smart POSIX and ADIOS libraries. The code and associated scripts just needed to be updated for the recent changes described above and to tailor its use on Garnet. Most of the modifications required for Smart POSIX were made directly to the library (see Section 2.1). However, it was necessary to relink the code to the new experimental library and to add a new library call to tell Smart POSIX not to quantize the floating-point data.

To use BEC with the ADIOS installed on Garnet, scripts were added to set up the environment before compilation and execution. Checks were also added in the Makefile to ensure that the proper environment has been loaded. Finally, ghost cell output was deemed unnecessary (since the cells could be reconstructed during post-processing if they were required), so the code and

ADIOS configuration files were adjusted to remove them. This also removes an extra communication step before writing, meaning that nearly pure I/O performance could be measured.

To provide fair comparisons, run scripts for the Smart POSIX, ADIOS POSIX, and ADIOS MPI-IO methods set optimal stripe counts and sizes for the run directories. The striping configurations followed the guide published by ERDC DSRC for Lustre filesystems. Specifically, both POSIX methods were given a stripe count of 1 (since each core writes 1 file) and the ADIOS MPI-IO method was given a large stripe count (80, since only a single file is used by all cores). This represents what any reasonable user would do when running a model with such output methods.

BEC was instrumented to use both the Catalyst and QIso co-processing libraries. Extra arrays were allocated as workspace for the co-processing libraries. A communication step was added for ghost cells to allow isosurfaces to be continuous across MPI task boundaries. Finally, the respective Catalyst/QIso calls were added to pass the data to the libraries and initiate co-processing every so many time steps.

Finally there were two minor, temporary modifications specifically for benchmarking and not necessary for regular use. The first of these was to include conditional compilation switches to easily enable/disable Smart POSIX, ADIOS, Catalyst, and QIso separately during compile time. Secondly, added as part of this PPE, were low-overhead timing routines to each write operation within each MPI task with an option to report the times to a log file.

2.4 Benchmark Setup

All of the I/O methods were benchmarked using six different job sizes. The Co-processing methods were benchmarked using the first five of the six. The job sizes were as follows:

- **Small:** 1728 cores, 1200^3 data points, 38.6 GB (per time step)
- **Medium:** 4096 cores, 2048^3 data points, 192 GB (per time step)
- **Large:** 10240 cores, 3040^3 data points, 628 GB (per time step)
- **Huge:** 46656 cores, 4860^3 data points, 2.50 TB (per time step)
- **Giant:** 92160 cores, 6000^3 data points, 4.71 TB (per time step)
- **Hero:** 150000 cores, 7200^3 data points, 8.15 TB (per time step)

When increasing the core counts for each job size, the number of grid data points is increased proportionally. This is known as *weak scaling*, using the extra cores to compute a larger solution rather than attempting to compute the same size solution faster. In the case of the BEC model, this does not increase the physical size of the domain; instead, it increases the resolution and thereby the fidelity of the model physics. This resolution increase is illustrated in Figure 2.

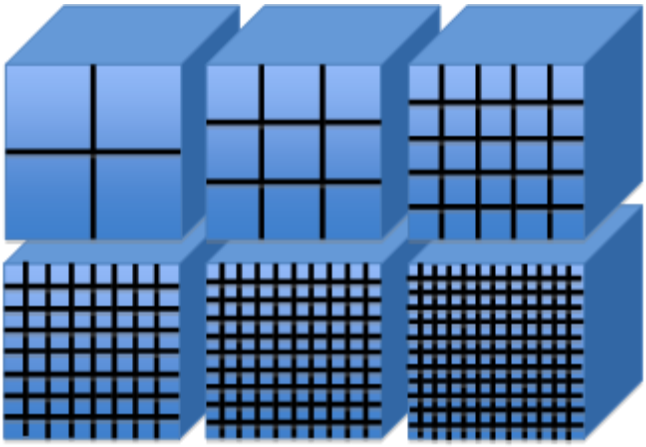


Figure 2. Diagram of the relative grid resolutions ranging from Small to Hero.

BEC computes some configurable number of time steps before writing output or co-processing. This number is tunable in the code and was set to write every 25 time steps for these studies. In some cases, if unexpected timing results were obtained, the job was run again to be sure that there were no system problems.

For each given benchmark run, timings from every MPI task were logged. A timer in each task and at every output time step, was started just before the I/O began and stopped when the I/O was finished in that task. Based on these timings, one can compute both mean and minimum throughputs for all tasks. The *mean throughput* is total data size divided by the average time for all task timings. The *minimum throughput* is the total data size divided by the *maximum* time for all task timings. Since there is a mean and minimum throughput computed for every output time step, the time step results are averaged together. Both mean and minimum throughputs are presented in this study, because the importance of the type of throughput will depend upon how a given user application integrates parallel I/O into the code. The minimum throughput represents the usual case that less advanced users will encounter, where no MPI task performs calculations while any task is doing I/O. Depending upon how well user codes overlap operations with the I/O, the mean throughput is an optimistic prediction of I/O performance. User code I/O performance will often lie somewhere in between the minimum and mean throughput.

Some of the I/O and co-processing methods seem to require some time to initialize. Their times for the first output could be significantly larger than subsequent time steps. Those timings were excluded from the minimum and mean throughput results. Instead, they were collected as separate *initialization throughput* timings and are presented separately in this report. Models that only write a few times (e.g., 1 or 2 restart file sets) may be strongly affected by any initialization throughput penalty. For models that write output at many time steps, the initialization penalty will likely average out.

Co-processing benchmarks were treated the same as I/O. That is, every MPI task timing was logged from the start of the co-processing activity to the end for each time step. A minimum throughput was computed based on the maximum time for all task timings. The mean throughput was not included because it is less likely to be able to overlap computation with the co-processing in any standard way (e.g., without specialized hardware). There were two planned configurations for Catalyst: Extract and Render. Catalyst Extract would compute isosurfaces

then write the surface geometry out to one file per core per time step. Catalyst Render would render the isosurfaces into a single image per time step. QIso only has a Render option.

3. RESULTS

Figure 3 and Figure 4 present the minimum and mean throughputs, respectively, for all five I/O methods at all 6 job sizes. Note the change in scale along the y-axis (throughput) from Figure 3 to Figure 4. As expected, the mean throughput can be significantly faster. Figure 5 similarly presents the initialization throughput. Its y-axis scale changes also, but notice that some methods get particularly slower. Figure 6 shows the minimum *effective throughput* for the Co-processing methods as compared the fastest of all the I/O methods at each job size. This is referred to as "effective" throughput because there is no actual I/O, but the co-processing is measured as if there were so that it can be compared to I/O performance. Note the large change in y-axis values.

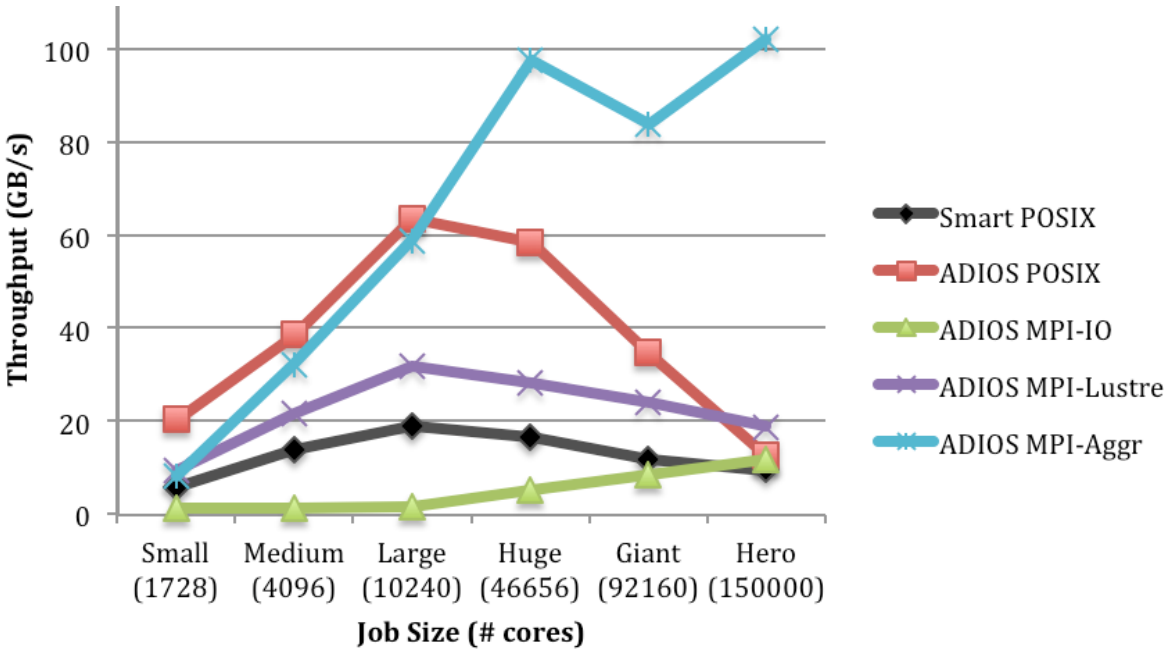


Figure 3. Minimum throughput results for the 5 I/O methods and 6 job sizes.

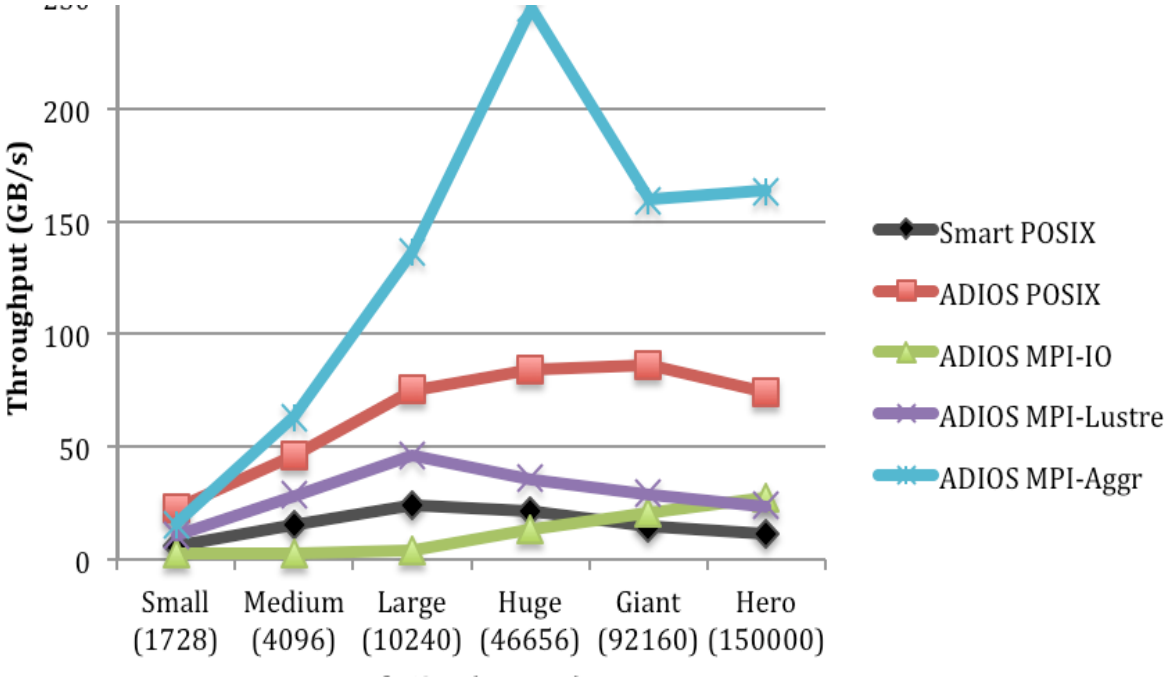


Figure 4. Mean throughput results for the 5 I/O methods and 6 job sizes.

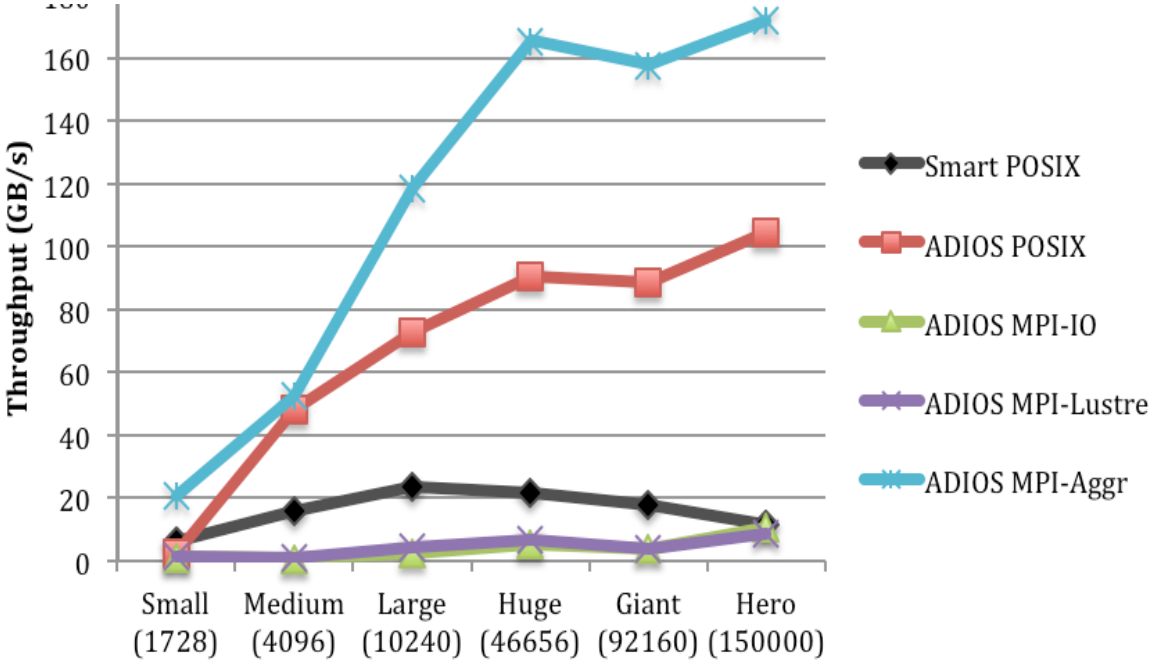


Figure 5. Initialization throughput results for the 5 I/O methods at 6 job sizes.

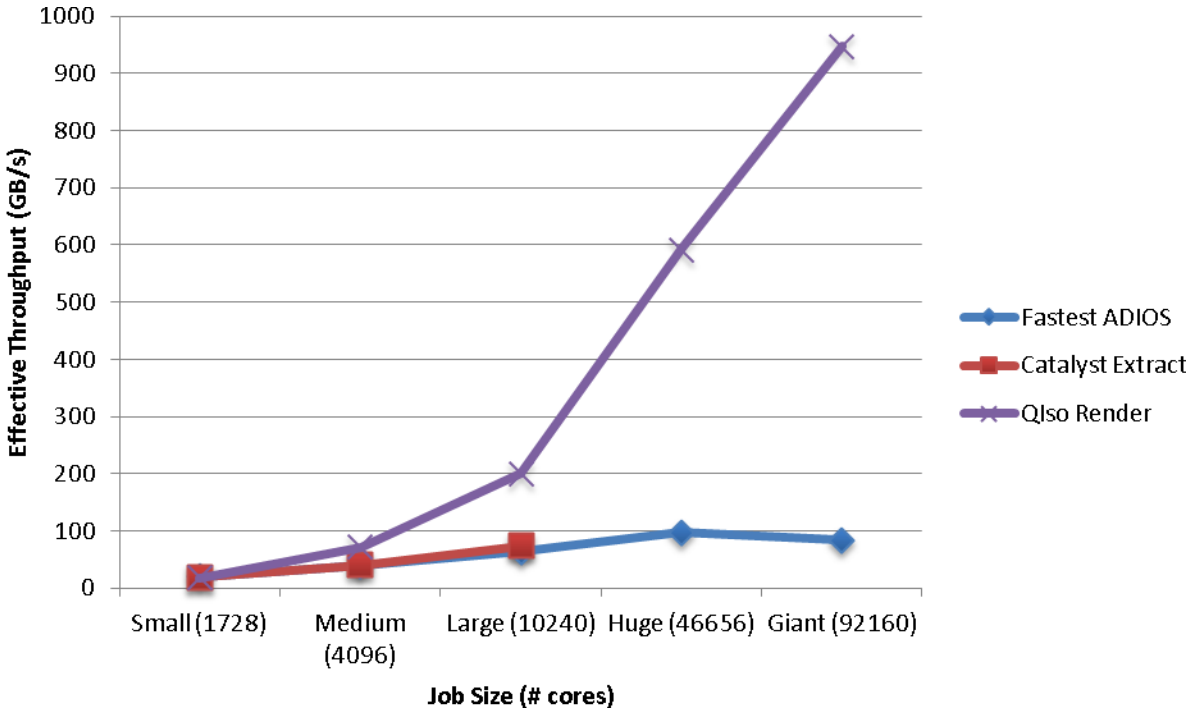


Figure 6. Effective throughput for the two Co-processing methods as compared to fastest I/O methods

ADIOS MPI-Aggr is the fastest I/O method for large scales (>10,000 cores), based on both minimum (Figure 3) and mean (Figure 4) throughput. For very large scales (>40,000 cores), it is an order of magnitude faster than other methods. This agrees with the ADIOS User Manual which states that the method is intended for scaling at 10,000 or more cores. When considering mean throughput, MPI-Aggr becomes the fastest approach at the medium scale (<4,000 cores). MPI-Aggr also has no apparent penalty during initialization, giving it the fastest initialization throughput (Figure 5) of all I/O methods at all scales.

ADIOS POSIX is the fastest method for small to moderate scales (<10,000 cores) based on minimum throughput. It is also the fastest in mean throughput at small scales (<4,000 cores). However, its performance begins to drop dramatically at very large scales (>40,000 cores). It also takes a performance hit during initialization at small scales.

Smart POSIX has similar scaling properties as ADIOS POSIX; i.e., its performance improves when increasing the number of cores until about 10,000 cores, where it begins to drop. This is not surprising, given that both methods write one file per core. However, at nearly every scale, it is a constant factor slower (roughly 1/3rd) than ADIOS POSIX, most likely due to ADIOS POSIX's buffering approach. It does seem to have no initialization penalty, but its overall speed negates any advantage that may provide.

ADIOS MPI-IO is the worst performing method overall. In minimum throughput, it is the worst method at all scales except at 150,000 cores. In mean throughput, it is the worst until roughly 90,000 cores. An interesting property though is that it begins to scale almost linearly after roughly 10,000 cores. Its starting point is so slow that this near-linear scaling is "too little, too late," but it would be interesting to see how the scaling progresses to many hundreds of

thousands of cores. ADIOS MPI-IO also has an initialization penalty, but it's small relative to its overall (poor) speed.

ADIOS MPI-Lustre performs moderately overall in terms of both mean and minimum throughput. However, it appears to achieve peak performance at the large (10,000) scale and then begins to lose performance moderately. By 150,000 cores, it is not much faster than the other slower methods (i.e., all but MPI-Aggr) in minimum throughput, and is even slower than MPI-IO in terms of mean throughput. The reason for this drop in performance at the largest scales is not known. MPI-Lustre also seems to suffer from the same initialization penalty as MPI-IO. In fact, their initialization throughputs are almost identical. However, since MPI-Lustre is overall significantly faster than MPI-IO, this means that the first time step will output much slower relative to any subsequent time steps.

There were problems compiling Catalyst on the latest Cray operating systems. So, we compiled and benchmarked it on Spirit (SGI ICE X at AFRL DSRC). Even on Spirit, the rendering component would not work, so we only have results for Catalyst Extract. And due to job size limitations, we only have results on Catalyst Extract for up to 10,240 cores. Also, due to delays caused by this, we could not complete any of the co-processing runs at the 150,000 core scale. Nonetheless, we obtained useful results up to 92,160 cores.

As shown in Figure 6, the Catalyst Extract option is not much faster than the fastest IO/ADIOS methods. We expect this is because the extraction still generates some amount of visualization data on nearly every core, which must then be written to one file per core. Each write operation, regardless of how small, will still require one file system metadata request per core when opening the files. This is effectively not much different than the Smart/ADIOS POSIX I/O methods. To improve performance, the Extract method's output routine would have to be rewritten to aggregate geometry to a smaller set of MPI tasks, in much the same approach that ADIOS MPI_Aggr uses, but with less overall data.

QIso Render outperformed all methods significantly (except at the smallest scale). This is because it is virtually entirely composed of calculations, just like the user code. It scales nearly linearly because isosurfaces are easily parallelized and the parallel compositing to generate the final image is a simple reduction operation. Other visualization algorithms may or may not scale as well, but it is expected that using isosurfaces with Catalyst Render would have similar performance.

4. SUMMARY

ADIOS is the best library for pure I/O performance. In particular, ADIOS POSIX works best for small scales, and MPI-Aggr for larger scales. However, other considerations may also factor in, where output files are concerned. MPI-Aggr outputs a single file per file system OST (with 696 OST's on Garnet). This is not as convenient as a single output file, but it is considerably better than one file per MPI task. That said, a single output file (i.e., from MPI-IO or MPI-Lustre) could potentially be extraordinarily large, which is not ideal for archival storage. Similarly, many small files (from POSIX methods) are not ideal for storage either. Having a modest number of moderately sized files tends to best for archiving, so this may be yet another attractive reason to use MPI-Aggr. Finally, one final annoyance of POSIX methods at large scales is that

hundreds of thousands of files can make user file management operations in a scratch directory very slow.

Using an extraction method for co-processing requires parallel I/O and, so far, is not much faster than simply writing out the entire data set. However, using a render method for co-processing reduces the entire process into a single image on a single core, which writes in negligible time. Overall, render-style co-processing is at least an order of magnitude faster than any approach using I/O, and even potentially scales up with the user code itself. This means that one could use render-style co-processing at frequent time step intervals. Figure 7 illustrates how a difference in output temporal resolution can provide a better understanding of how the physical properties can develop over a short time period. The images with black boxes indicate the time step output visualizations that one would see at every 400 time steps (common for the BEC code, depending upon size). The images surrounded by green dashed boxes show every 25 time steps. In this case, once can see much more precisely how the features developed and changed.

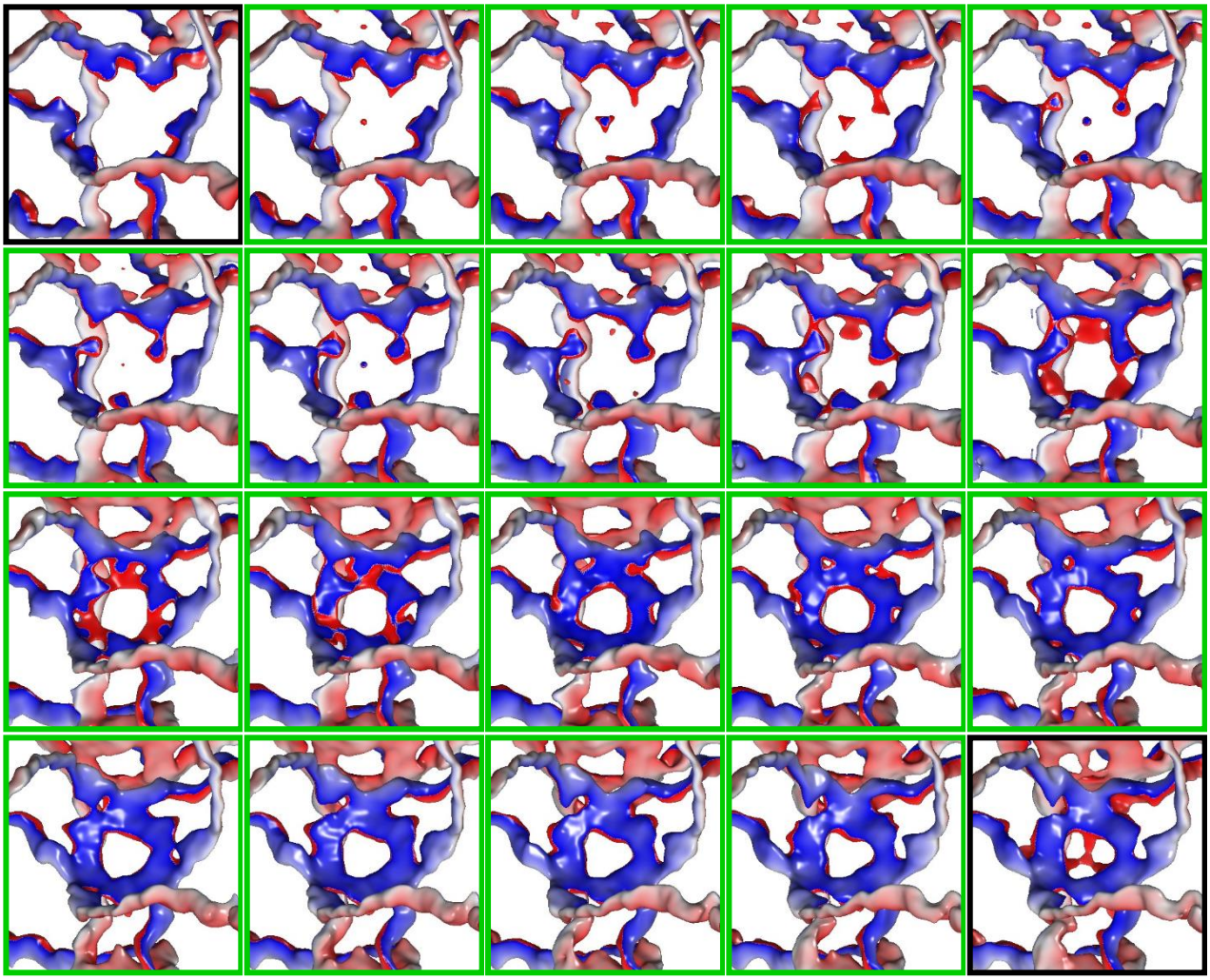


Figure 7. Example BEC model output visualizations from QIso over 400 time steps total. The black boxes indicate a usual coarse output time resolution (every 400 time steps). The green boxes indicate fine time resolution (every 25 time steps).

Finally, an often neglected topic is that render-style co-processing also completes the post-processing step automatically. So, not only could the computation be completed faster and provide better time resolution output, the total time-to-solution should also be reduced.

When deciding upon co-processing options, Catalyst is usually the most attractive because of the variety of extraction, visualization, and rendering options readily available. When changing the desired visualization results, exporting different configuration scripts from the ParaView GUI is not difficult, especially after some training. It also provides both Render and Extract options. Once the Catalyst code becomes more stable, we will likely recommend it to most users. Simpler approaches like QIso are very easy to use; the instrumentation calls are even easier than Catalyst's. However, QIso provides limited options and is difficult to modify. To add another feature (e.g., a new visualization method or an Extract option), the C++ code must be modified. Moreover, the programmer making the modification must be familiar with implementing visualization algorithms from scratch.

Overall, ADIOS MPI-Aggr is an efficient method with a reasonable output format and seems to be the best method for I/O in general. ADIOS POSIX can be faster at smaller scales, but creates file management difficulties at any scale. However, if one can utilize co-processing in some form, it can often provide unparalleled speedup, file management advantages, and finer results.

5. REFERENCES

[1] N. Podhorszki, Q. Liu, J. Logan, H. Abbasi, J.Y. Choi, and S. Klasky, "ADIOS 1.5.0 User's Manual." U.S. Department of Energy, Office of Science, Dec. 2012, pp. 10-105.

[2] J. Yepez and B. Boghosian, "An efficient and accurate quantum lattice-gas model for the many-body Schrodinger wave equation", Computer Phys. Comm. 146, 2002, 2080-2094.

[3] G. Vahala, J. Yepez, L. Vahala, M. Soe, B. Zhang, and S. Ziegeler, "Poincare recurrence and spectral cascades in three-dimensional quantum turbulence", Phys. Rev. E. 84, 046713, 2011, pp. 1-17.

[4] K. Moreland, N. Fabian, P. Marion, and B. Geveci, "Visualization on Supercomputing Platform Level II ASC Milestone (3537-1B) Results from Sandia", Sandia Report SAND2010-6118, Sandia National Laboratories, 2010, pp. 7-21.

[5] A. Bauer, B. Geveci, and W. Schroeder, "The ParaView Catalyst User's Guide 1.0", Kitware Inc., 2013, pp. 2-67.